Supporting Data Consistency in Concurrent Process Execution With Assurance Points and Invariants

Susan D. Urban[1], Andrew Courter[2], Le Gao[2]
[1]Department of Industrial Engineering
[2]Department of Computer Science
Texas Tech University
Lubbock, TX
{susan.urban | s.courter | le.gao}@ttu.edu

Mary Shuman
Department of Computer Science
University of North Carolina, Charlotte
Charlotte, NC
mary.shuman@gmail.com

***Abstract*—This research has developed the concept of invariants for monitoring data in a service-oriented environment that allows concurrent data accessibility with relaxed isolation. The invariant approach is an extension of the assurance point concept, where an assurance point is a logical and physical checkpoint that is used to store critical data values and to check pre and post conditions related to service execution. Invariants provide a stronger way of monitoring constraints and guaranteeing that a condition holds for a specific duration of execution as defined by starting and ending assurance points, using the change notification capabilities of Delta-Enabled Grid Services. This paper outlines the specification of invariants as well as the invariant monitoring system for activating invariants, evaluating and re-evaluating invariant conditions, and deactivating invariants. The system is supported by an invariant evaluation web service that uses materialized views for more efficient re-evaluation of invariant conditions. The research includes a performance analysis of the invariant evaluation Web Service. The strength of the invariant technique is that it provides a way to monitor data consistency in an environment where the coordinated locking of data items across multiple service executions is not possible.**

Keywords: web services, invariants, data consistency, data monitoring, concurrent data access

I. Introduction

In service-oriented computing, business processes are composed by executing Web Services [12]. Although each Web Service is autonomous and self-contained, composing business processes and achieving a correct global solution is a difficult and sometimes error-prone task, especially in the context of concurrently executing processes that access shared data.

In traditional distributed transaction systems, the two-phase commit (2PC) protocol [5] has been used to support the ACID properties of atomicity, consistency, isolation, and durability. In service-oriented computing, however, it is generally not feasible to support ACID properties by coordinating the commit time of all services that are part of a global process because of the loosely-coupled, autonomous, and heterogeneous nature of services. Moreover, in traditional transaction processing, the concept of serializability is supported by using locking protocols [5]. In service-oriented computing, however, it is not practical to require constituent services to lock data for the entire duration of a global process. This is especially true for long-running processes, causing processes to execute using a relaxed form of isolation in between service executions. As a result, the correctness of a process might be affected by another concurrently running process if both processes are accessing shared data. Insuring the consistency of data in a service-oriented environment with relaxed isolation is a challenging task.

This paper presents the concept of *invariants* for monitoring data in a service-oriented environment that allows concurrent data accessibility with relaxed isolation. The invariant technique is an extension to the concept of an *assurance point* (AP) as defined in [14, 19]. An AP is a logical checkpoint created in between the service calls of a process, defining a named point that can be used to store critical data values, to express a post-condition for completed services, and to express a precondition for the next service to execute. APs are also used as intermediate rollback points to assist with backward and forward recovery actions when process failure occurs.

An invariant is a condition that must remain true during process execution in between two different APs. An invariant is specifically designed for use in processes where 1) isolation of data changes in between service executions cannot be guaranteed (i.e., critical data items cannot be locked across multiple service executions), and 2) it is critical to monitor constraints for the data items that cannot be locked. The data monitoring functionality provided by the work with Delta-Enabled Grid Services (DEGS) [2, 18] makes it possible to declare and monitor invariant conditions.

This research has involved the specification of invariant conditions as well as the design and development of a prototype invariant monitoring system. When a process declares an invariant condition, if a concurrent process modifies a data item of interest in an invariant condition, the process that activated the invariant is notified by the monitoring system built on top of Delta-Enabled Grid Services. If the invariant condition is violated during the specified execution period, the process can invoke recovery procedures as defined in [19]. The monitoring system includes the design of a Web Service for evaluating invariants. Since an invariant may need to be evaluated several times between the starting and ending APs of an invariant, the invariant evaluation Web Service was designed to make use of materialized views for more efficient re-evaluation of invariant conditions [15]. The research includes a performance analysis of the invariant evaluation Web Service, illustrating the benefits of using materialized views.

Whereas the original work with APs allows data consistency conditions to be checked at specific points in the execution, invariants provide a stronger way of monitoring constraints to determine if a condition holds for a specific duration of execution without the use of locking.

The remainder of this paper is organized as follow. After outlining related work in Section II, Section III provides an overview of the Delta-Enabled Grid Services and Assurance Point concepts that provide the basis for supporting the invariant approach. Section IV presents an overview of the design and functionality of the Invariant Monitoring System. A prototype of the Invariant Monitoring System is described in Section V, followed by a discussion of the testing and evaluation results in Section VI. The paper concludes in Section VII with a summary and discussion of future research.

II. Related work

Past research with transactional workflows has investigated the need to relax ACID properties for long running workflow activities [21]. The Saga transaction model was proposed as a base model for long-running activities and defines a chain of transactions as a unit of control [6]. The Saga model relaxes the requirement of the entire transaction as an atomic action by releasing a resource before it completes without sacrificing the consistency of the database. Models similar to the Saga model are called Advanced Transaction Models (ATMs). A model that has been used to define and study transactional workflow is the ConTracts model [20].

Several new approaches for addressing transactional issues have been defined in the context of web services. A goal of the Promises project [7] is to make sure that certain values are not overwritten or changed by concurrently executing Web Services. A promise is an agreement between a client application and a service or promise maker. The promise maker guarantees that some set of conditions will be maintained over a set of resources for a specified period of time. Another similar method to temporarily perform physical and logical locks over data in a concurrent environment is the reservation-based approach [22]. The reservation-based approach reserves resources that meet the criteria of what the Web Service has requested. Only the required amount of a resource is reserved, rather than locking the database record or the entire resource for an extended period of time.

Transactional Attitudes are used as a framework to handle the transactional reliability issue in Web Services. Transactional Attitudes establish a separation of transactional properties from other aspects of a service description. In [13], the WSTx framework uses transactional attitudes that make Web Service providers declare their individual transactional capabilities and semantics, and Web Service clients declare their transactional requirements.

The work in [1] uses monitoring rules woven inside of a WS-BPEL process to dynamically control the execution during runtime. The monitoring rules are annotated in the source code using assertion languages, such as Anna (Annotated Ada) [11] and JML (Java Modeling Language) [10] . User-defined constraints are blended with the WS-BPEL process at deployment time and are defined externally to allow separation of the different functionalities.

The work presented in [3] uses aspect-oriented concepts to address the modularity issues in workflow languages. A prototype extension to BPEL using aspect-oriented workflow concepts (AO4BPEL) [4] was developed to validate their work. A well known aspect-oriented programming language, AspectJ [9], uses three key concepts: join points, pointcuts, and advice, to support the aspect portion of the aspect-oriented workflows and AO4BPEL described in [3].

Using the techniques describes in this section, constraint conditions cannot be monitored during a specific execution duration. The focus of the research presented in this paper is to present a system to extend the Assurance Point architecture to allow monitoring of critical data conditions during specific execution periods in a process. Providing this capability allows a more optimistic approach to concurrent process execution but also allows data inconsistencies to be more quickly recognized.

III. Background for the use of Invariants

Before presenting the Invariant Monitoring System, it is first necessary to provide background on Delta-Enabled Grid Services (DEGS) and Assurance Points. DEGS support the ability to invoke constraint checking actions after a change in the source database. Assurance Points provide the framework inside of a business process to activate invariant conditions and to define the time frame for condition monitoring.

A. *Delta-Enabled Grid Services*

A DEGS is a Grid Service that has been enhanced with an interface that stores the incremental data changes, or *deltas*, that are associated with service execution in the context of globally executing processes. A DEGS uses the OGSA-DAI Grid Data Service for database interaction. The DEGS functionality was originally defined in [2] and has been used to determine data dependencies among concurrently executing processes to support process recovery actions [18].

Using the DEGS approach, a database captures deltas using capabilities provided by most commercial database systems. The work in [2, 18] experimented with triggers and with the use of Oracle Streams as a way to capture data changes. Oracle Streams is a feature that monitors database redo logs for changes and publishes these changes to a queue to be used for data sharing [16].

Using the DEGS approach, when a change to the source database is made by a Grid service, the delta is captured and inserted into a delta repository. The delta repository has a separate table for inserts, deletes, and updates to each source database table, allowing information about each type of change to be kept separate. Additionally, a table mapping each delta to information about the Grid service that made the change is kept.

A Java stored procedure deployed in the source database is automatically called to notify a listening Grid service that there are new deltas in the table that was just modified. The listening Grid service then looks for new deltas in delta repository tables. These deltas are compiled into an XML format and then relayed to any other system that has registered to receive the delta information, such as the Invariant Monitoring System described in this paper.

B. *Service Composition and Recovery with APs*

As described in [19], an Assurance Point (AP) is a logical and physical checkpoint for storing data and using rules, known as *integration rules* (IRs), to check pre and post conditions at critical points in the execution of a process. Given that concurrent processes do not execute as traditional transactions in service-oriented environments, inserting APs at critical points in a process is important for checking consistency constraints and potentially reducing the risk of failure or inconsistent data.

An AP can also be used as a rollback point for backward recovery. Three different forms of backward recovery are described in [19], with the different forms supporting either full backward recovery or a combination of backward and forward recovery. APRetry is used when the running process needs to be backward recovered to a previously-executed AP. APRollback is used when the overall process has more severe errors and must be recovered back to the beginning of the process. APCascadedContingency is a hierarchical backward recovery that continues to compensate nested processes, checking each AP that is encountered for a possible contingent procedure that can be used to correct an execution error.

The most basic use of an AP together with integration rules is shown in Figure 1, which illustrates three composite groups (i.e., code segments that invoke services) and an AP between each composite group. The shaded box on the right shows the functionality of an AP using AP2 as an example. When AP2 is reached, the post-condition rule, the pre-condition rule, and any conditional rules are checked sequentially. If the post-condition or the pre-condition is violated, then a recovery action is invoked. If the pre and post conditions are not violated, then the AP will invoke any conditional rules to check additional, application-oriented conditions.

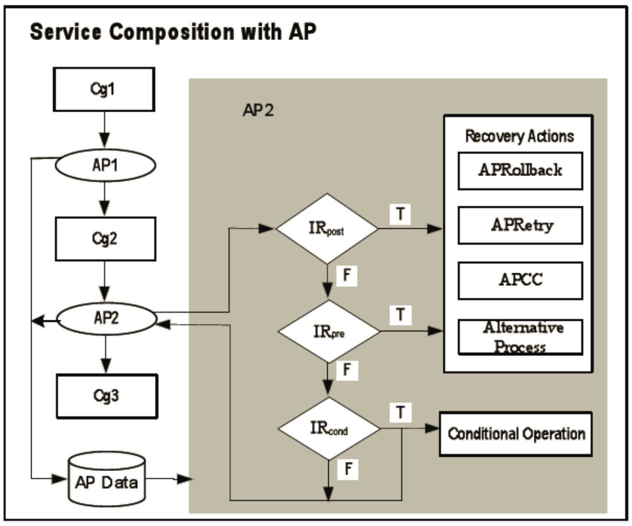


Figure 1 Assurance Points and Integration Rules [14]

The Invariant Monitoring System extends the functionality of Assurance Points by adding an additional invariant rule, where an invariant rule allows the specification of a condition that can be monitored in between two AP occurrences. IRs for pre and post conditions can check conditions at certain points in the business process but cannot make sure that a condition holds for a specified period of time. The Invariant Monitoring System provides the capability to monitor critical data conditions in between APs, supporting concurrent activity but allowing a process to be notified if a critical data condition is violated.

IV. INVARIANT MONITORING SYSTEM

This section presents an overview of the functionality of the Invariant Monitoring System. The format for invariant rule specification is then presented using two examples that will be used throughout the remainder of the paper.

A. *Overview*

Using the invariant technique, a process declares an invariant condition when it reaches a specific AP in the process execution, also declaring an ending AP for monitoring of the invariant condition. When a concurrent process modifies a data item of interest in an invariant condition, the process that activated the invariant is notified by a monitoring system built on top of Delta-Enabled Grid Services. If the invariant condition is violated during the specified execution period, the process can invoke the recovery procedures defined in the previous section.

An invariant definition has an identifier, two AP specifications (AP_s as a starting AP and and AP_e as an ending AP), and optional parameters that are necessary in the condition specification. Once AP_s is reached, the invariant rule condition becomes active. The condition is specified as an SQL query. The condition is initially checked and the action is executed if the invariant condition is violated. If the invariant condition holds, the rule condition goes into monitoring mode using the DEGS capability. The condition monitoring continues until AP_e is reached or until the invariant condition is violated.

As shown in Figure 2, when an invariant condition goes into monitoring mode, the data items of interest in the invariant condition are registered with a monitoring service. The monitoring service subscribes to the DEGSs that contain the relevant data items referenced in an invariant. For example, if the condition to be monitored is a + b > 10, then the relevant DEGS will notify the service of any changes to a or to b by concurrent processes. Any deltas that are forwarded to the monitoring service will cause the invariant condition to be rechecked. As long as the condition still holds, then there is no interference among the concurrent process executions. If the condition is violated, then the recovery action of the invariant rule will be executed.

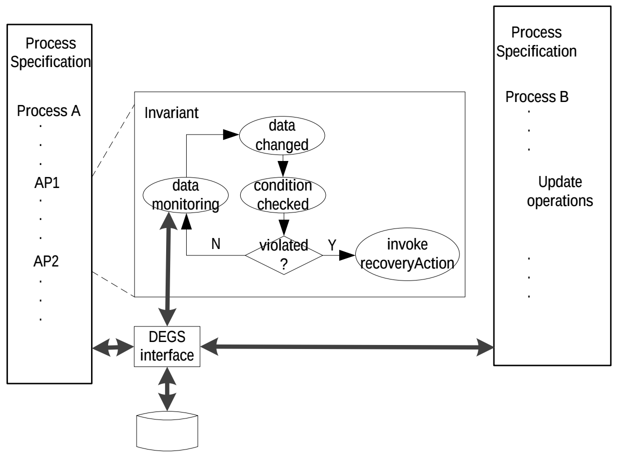


Figure 2 Invariant System

B. *Invariant Specification*

Assurance Points uses integration rules in the Event-Condition-Action (ECA) format to define the different types of integration rules. These ECA rules are based on previous work with using integration rules to interconnect software components [8, 17].

Each invariant begins with a create rule statement that defines an invariant identifier. The event component of the rule identifies the starting AP as well as the ending AP and any parameters needed for the rule condition specification. In the condition section of the ECA rule structure, the condition is expressed as not exists (select * from ...), where the select statement returns the tuples that satisfy the invariant condition. If the select statement returns tuples that satisfy the condition, then not exists evaluates to false and no recovery action is triggered. However, if the SQL condition returns no tuples, then not exists will return true, indicating that the invariant condition is not satisfied. In this case, the process is notified and the recovery procedure in the action is invoked.

Hotel Room Reservation Monitoring Example

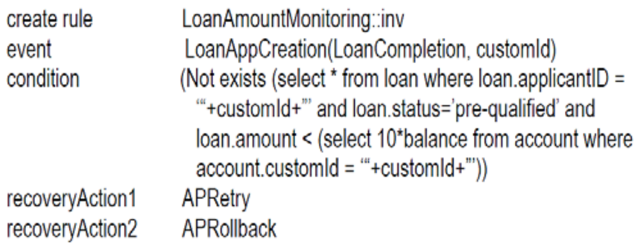
Figure 3 provides an example of an invariant for a travel planning process, where the process is scoping out available hotel and airline options before finalizing the plans. The full details of the process are not presented here, but the invariant is triggered when the process reaches the BeginTravelPlanning AP as specified in the EVENT component of the invariant rule. The first parameter of the event specifies that the invariant is deactivated when the process reaches the ReadyToBook AP. The invariant condition checks a specific hotel for the availability of a seaside room that is less than a specified price, where the hotelID and price are passed as additional parameters from the BeginTravelPlanning AP. Expression of the invariant allows the process to continue checking the availability of other travel options, such as airline reservations, but to be notified if the room availability changes. If the process reaches the ReadyToBook AP and the desired room type and price are still available, then the process continues past the ReadyToBook AP, making the appropriate reservations after deactivating the HotelRoomMonitoring invariant. If at anytime between the BeginTravelPlanning AP and the ReadyToBook AP the room is no longer available, the invariant monitoring system will notify the process instance that owns the invariant condition.

```
create rule         HotelRoom Monitoring :: inv
event               BeginTravelPlanning  (ReadyToBook,
                    hotelID, price)
condition           (Not exists (select * from Rooms R
                    where R.roomPrice <  '"+ price +"'and
                    R.roomType  = 'seaview' and
                    R.hid = '"+hotelID+"'))
recoveryAction1     APRetry
recoveryAction2     APRollback
```

Figure 3 Invariant for a Hotel Room Reservation Request

Bank Loan Application Monitoring Example

As another example, consider the invariant in Figure 4, where the LoanAmountMonitoring invariant is to be monitored between the LoanAppCreation AP (i.e., the starting AP for the monitoring process) and the LoanCompletion AP (i.e., the ending AP for the monitoring process). The process represents a loan approval process, where the process is creating a loan application for a customer at a bank that already has an account at that bank. Figure 4 shows an invariant that is activated when the LoanAppCreation AP is reached and checks to make sure the loan applicant has a tenth of the requested loan amount in the account, where the customerId is passed as a parameter from the LoanAppCreation AP. The monitoring process is started if the condition is satisfied. If the process reaches the LoanCompletion AP and the applicant's account balance still meets the necessary criteria, then the process continues past the LoanCompletion AP, completing the loan application after deactivating the LoanAmountMonitoring invariant. If at anytime between the LoanAppCreation AP and the LoanCompletion AP, the applicant's account balance falls below the necessary criteria, the invariant monitoring system will notify the process, which will execute the recovery action.



create rule	LoanAmountMonitoring::inv
event	LoanAppCreation(LoanCompletion, customId)
condition	(Not exists (select * from loan where loan.applicantID = '"+customId+"' and loan.status='pre-qualified' and loan.amount < (select 10*balance from account where account.customId = '"+customId+"'))
recoveryAction1	APRetry
recoveryAction2	APRollback

Figure 4 Invariant for a Bank Loan Approval Process

V. PROTOTYPE OF THE INVARIANT MONITORING SYSTEM

As part of our research, we have prototyped an execution environment to model the capability of monitoring invariants in between the APs of an executing process. This section outlines the relevant components of the invariant monitoring system.

A. *Registration of Invariants and Monitored Objects*

Invariant rules are parsed and processed to extract the SQL condition and the monitored objects from the invariant rule definition. Monitored objects are acquired from the SQL condition of an invariant by extracting the table names together with the attributes and relevant conditions. Changes to these extracted objects can affect the result of the query. The Invariant Monitoring System may need to re-evaluate the SQL condition when it detects a change in monitored objects.

As an example, consider the SQL query from Figure 4. The two tables in this query are the Loan table and the Account table. There are three conditions in the where clause of the outer SQL query associated with the Loan table. As a result, there are three monitored objects from this table: "applicantId = +customerId+", "status = 'pre-qualified'", and "amount < (select ...)". To simplify the monitored object related to the amount attribute, the object is converted into "amount < calc" since multiple tables cannot be analyzed during the delta filtering. The calc keyword is used to signify that this is a calculated value that must be re-evaluated. In the first condition, customerId is a parameterized value that is acquired from the parameters of the AP.

The Account table of the inner query has one condition in the where clause, "customerId = +customerId+", where customerId is a parameterized value. This query also illustrates a relevant monitored object in the select clause for the balance attribute of the Account table. Balance is identified as a calculated value since, if this attribute changes, it will change the output of the inner query and could potentially violate the invariant condition.

After parsing an invariant rule, an object structure is used to forward information about the invariant to an Invariant Agent, which validates the condition and registers the invariant and its list of monitored objects with the system if the condition is satisfied. Figure 5 shows a high level view of the relationship between the MonitoredObject table and the Invariants table in the Invariant Agent. As shown in Figure 5, there is a many-to-many relationship between MonitoredObjects and Invariants. If an invariant no longer needs to be monitored, then it is deactivated and deleted from the Invariants table. If the objects related to that invariant are not related to another invariant, they will also be removed.

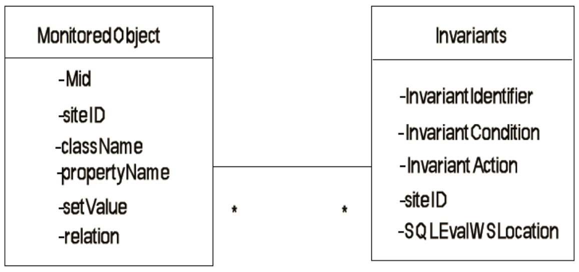


Figure 5 Invariants and Monitored Objects Table

B. *The Invariant Evaluation Web Service*

An important component of the Invariant Monitoring System is the Invariant Evaluation Web Service (Shuman, 2010). The Web Service is used to initially evaluate the SQL query of an invariant to determine if the condition is satisfied. Since the invariant may need to be re-evaluated several times between the starting and ending APs, the Web Service was designed to make use of materialized views to provide an efficient way of checking the invariant.

A materialized view is a database object that contains the results of a query. After populating a materialized view when an invariant is initially evaluated, the view is automatically updated after any table that is associated with the query is changed. In Oracle, this is referred to as the FAST refresh option. As a result, simply counting the number of tuples from the materialized view is faster and more efficient than re-executing the SQL query when an invariant must be re-evaluated. As long as the count is greater than zero, the constraint is still satisfied. An empty view indicates that the constraint is not satisfied.

Figure 6 illustrates the functionality of the Invariant Evaluation Web Service. After creating any necessary log files needed for the FAST refresh option, the Invariant Evaluation Web Service determines if the materialized view exists. If the view does not exist, the materialized view is populated by executing the query of the invariant. If the materialized view already exists, then the number of tuples is queried from the view instead of re-executing the query.

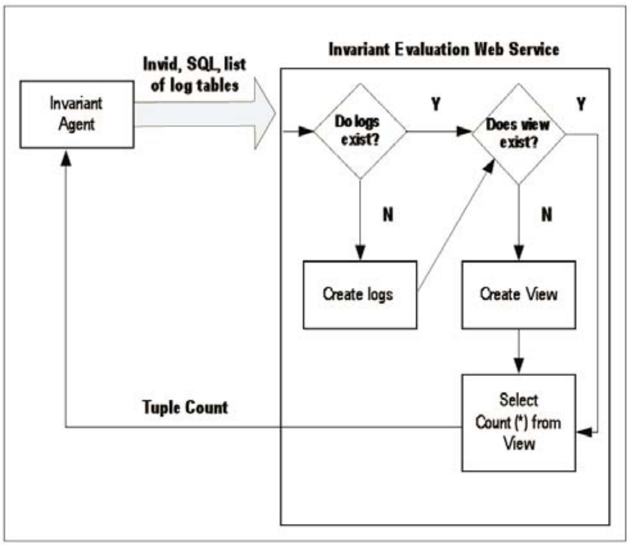

Invid, SQL, list of log tables
Invariant Evaluation Web Service
Invariant Agent
Do logs exist?
Y
Does view exist?
Y
N
N
Create logs
Create View
Select Count (*) from View
Tuple Count

Figure 6 Evaluation Web Service Functionality

C. The Delta Analysis and Filtering Process

The Delta Analysis agent of the Invariant Monitoring System invokes the filtering of delta information received from DEGS against the monitored objects. To support the delta filtering process, a storage container for the monitored objects is required. Figure 7 shows the Delta Analysis Agent (DAA) Invariant Storage Container, which consists of two hashtables. The first hash table is the table/attribute hashtable containing a vector of invariant identifiers that have monitored objects containing the same table/attribute combination as the key. For example, if an invariant is monitoring the price attribute in the orders table, then the key would be orders/price and the invariant identifier of that invariant would be inserted into the container of that key in the table/attribute hashtable. The second hashtable, or invariant hashtable, uses the invariant identifier as the key and relates that key to a container of monitored objects of that invariant. The first entry in the container contains information about the number of tuples that the last evaluation of the invariant found, the current number of violations found against that invariant identifier, and the invariant identifier. The rest of the container holds the monitored objects that are related to that invariant so that all conditions related to that invariant can be checked at the same time.

To process delta notifications, a delta filtering process was developed using two different algorithms, where one algorithm handles insert and delete operations and the other algorithm handles updates. In addition, each algorithm distinguishes between invariants that involve a single table and invariants that involve multiple tables.

To allow a more efficient method of determining when to re-evaluate an invariant that applies to a single table, a variable containing the number of tuples returned from the SQL query was introduced. Since all of the monitored objects are evaluated over a single table, the filtering process can use tuple counts to determine when an invariant is violated. If the number of tuples equals the number of violations found, then there are no more tuples left that satisfy the invariant condition and the Delta Analysis Agent can deduce that the condition has been violated without re-evaluating the condition.

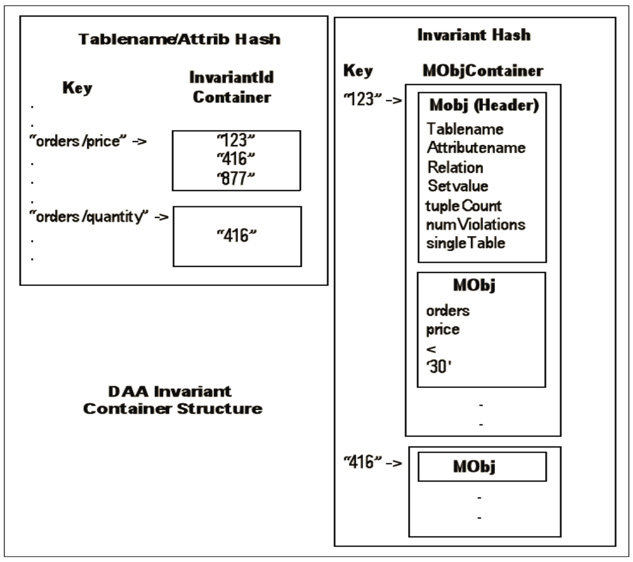


Figure 7 Delta Analysis Agent Storage Structure

Example 1: Single table insert
Invariant: "select r.price from room r where r.price < '30' and r.roomType = 'seaview' and r.hotelid = '234'"
Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (room, hotelid, =, '234')]
Number of Satisfying Tuples: 1
Discussion: If a tuple satisfying all of the monitored object conditions is inserted into the room table, then the number of tuples is incremented by one. If one of the monitored object conditions is not satisfied by the inserted tuple, then the number of tuples is not incremented (i.e., the inserted data has not affected the contents of the view). If the tuple that satisfies the invariant is deleted by an external process, then the number of violations will be incremented. Since the number of tuples will equal the number of violations, notification will be sent to the process monitoring the invariant condition. The invariant will be removed from the monitoring process, and the process will be informed of the violation.

When monitoring multiple tables, a threshold value is used instead of comparing the number of tuples and the number of violations found. Invariants that involve join conditions and, therefore multiple tables, require rechecking the invariant condition. A tuple from one table can join with multiple tuples from another table. As a result, an insert, delete, or update can cause multiple tuples to enter or leave the result of the invariant. Furthermore, depending on the number of tuples in the invariant result, these changes do not necessarily violate the invariant condition. It is not desirable, therefore, to check the invariant after each change to a relevant table. A threshold value is used as a way to periodically initiate a re-evaluation, where the threshold value is a percentage of the number of tuples that determines when to re-evaluate the invariant condition. This research has used a threshold value of 25% of the invariant tuples.

Example 2: Multiple table insert
Invariant: "select r.price from room r, hotel h where r.price < '30' and r.roomType = 'seaview' and r.hotelid = h.hotelid and h.state = 'Texas'"
Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (hotel, state, =, 'Texas')]
Number of Satisfying Tuples: 25
Threshold: 25%
Discussion: All inserts into multiple table invariants are ignored. Inserting tuples can potentially increase the size of the number of tuples that satisfy the invariant condition, but will not cause a violation. If seven tuples from the room table satisfying the invariant condition are deleted, then the number of violations will be incremented after each deletion. After the seventh deletion, the number of violations will be greater than the threshold (7 > .25*25). The invariant condition will then be re-evaluated. If tuples are found that satisfy the invariant condition, the invariant will update the number of tuples found in the view, reset the number of violations to zero, and continue monitoring. If the process continues and after another re-evaluation no more tuples are found, a notification will be sent to the process monitoring the invariant condition and the invariant will be removed.

VI. Testing and Evaluation

To evaluate the prototype of the Invariant Monitoring System, a testing environment and test cases were created and initialized. The primary focus of the evaluation was on the performance of the Invariant Evaluation Web Service to determine if the use of materialized views improves the performance of the re-evaluation process. The testing example used was the Hotel monitoring example, which involves the Hotel and Room tables, with DEGS created to monitor changes to all columns of each table. A process with Assurance Points was created for activating and deactivating different test invariants. Another concurrent process was also created to modify the monitored data in the source database.

Since re-evaluation occurs primarily in the context of multiple table invariants, the focus of the evaluation was on invariants that involve join conditions. The first test case involved changes ranging from satisfying the invariant condition to not satisfying the invariant condition. These updates triggered the invariant condition to be re-evaluated, but the test was design so that the invariant condition was still satisfied and, as a result, the invariant was not removed. The second test case updated all of the tuples with the changes ranging from satisfying the invariant condition to not satisfying the invariant condition. This test group was designed so that the invariant was violated and, as a result, monitoring of the invariant was removed (the evaluate the time associated with removal of the monitored invariant).

Table 1 describes different measurements that were taken and the times associated with each measurement. The measurements taken include:

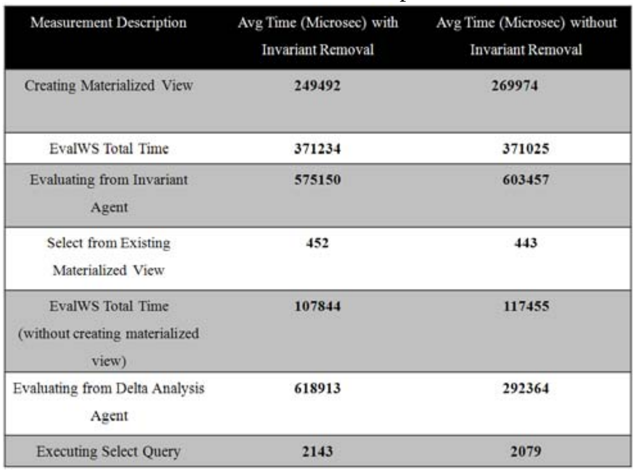
- The time for creating the materialized view, where the time includes creating the view and extracting the number of tuples from the newly created view.
- The total time of the Invariant Evaluation Web Service, which includes checking and creating any logs, and either creating and querying from the materialized view or just querying tuple counts from the materialized view if it already exists.
- The time to evaluate the invariant from the Invariant Agent, which includes the time to call and receive feedback from the re-evaluation function in the Invariant Agent for evaluating the invariant condition the first time,
- The time to evaluate the invariant from the Delta Analysis Agent, which is the time is takes to call and receive feedback from the re-evaluation function in the Invariant Agent. The time taken can also include the time it takes to remove the invariant condition if there are no more tuples in the view.
- The time to select tuple counts from the materialized view.
- The time to directly execute the SQL query of an invariant instead of creating a materialized view.

An average time in microseconds was recorded for all measurements. During testing, the Oracle database used had at least 100 tuples that satisfied the invariant condition on the initial evaluation. The machine used for testing was a Dell Precision T3400 with 2.99GHz Intel Core 2 Extreme processor and 4Gb of RAM, running Microsoft Windows XP Professional x64 Edition.

Creating the materialized view and the total evaluation time was about the same in both multiple table test cases. The values that are significantly different in the test cases are the times for evaluating from the Delta Analysis Agent. This time difference is because the invariant condition is completely removed in the case with invariant removal before returning back from the Delta Analysis Agent evaluation, indicating the time required for deactivation of the invariant.

A key observation from Table 1 is that the time difference between creating the materialized view and selecting tuple counts from an existing materialized view is significantly different. The time it takes to query tuple counts from an existing materialized view is also much less than the time required to repeatedly re-execute the invariant query. If a process is long running between the starting and ending APs of an invariant and might potentially re-execute the SQL query of the invariant often, then creating the materialized view is beneficial. Otherwise, directly re-evaluating the query is a better choice for shorter process to avoid the overhead of establishing the materialized view.

Table 1 Performance of Multiple Test Cases



Measurement Description	Avg Time (Microsec) with Invariant Removal	Avg Time (Microsec) without Invariant Removal
Creating Materialized View	**249492**	**269974**
EvalWS Total Time	**371234**	**371025**
Evaluating from Invariant Agent	**575150**	**603457**
Select from Existing Materialized View	**452**	**443**
EvalWS Total Time (without creating materialized view)	**107844**	**117455**
Evaluating from Delta Analysis Agent	**618913**	**292364**
Executing Select Query	**2143**	**2079**

VII. Summary and Future Work

This paper has introduced the design of an Invariant Monitoring System that is capable of monitoring data constraint conditions in a process, using the AP concept from [19] as a way to define the monitoring period. A web service was developed that makes use of materialized views, an invoked by delta filtering algorithms, as a way to improve the efficiency of the invariant re-evaluation process. The strength of the invariant technique is that it provides a way to monitor data consistency in an environment where the coordinated locking of data items across multiple service executions is not possible, thus providing better support for the reliability user-defined correctness conditions among concurrent processes.

Future research is needed to more accurately define the threshold that is used to determine when to invoke the invariant evaluation web service. The web service could also be enhanced to make dynamic decisions regarding the use of materialized views vs. direct re-execution of the invariant query. More efficient methods of checking invariant conditions should also be investigated. For example, using DEGS to directly monitor the materialized views instead of the delta repository tables could provide a more immediate solution to monitoring multiple tables.

Acknowledgment

This research has been supported by NSF Grants CCF-0820152 and CNS-1005212. Opinions, findings, conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of NSF.

References

[1] L. Baresi, and S. Guinea, “Towards dynamic monitoring of WS-BPEL processes,” Service-Oriented Computing-ICSOC 2005, pp. 269-282, 2005.

[2] L. Blake, The Design and Implementation of Delta-enabled Grid Services: MS Thesis, Arizona State University, 2006.

[3] A. Charfi, and M. Mezini, “Aspect-oriented workflow languages,” On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, pp. 183-200, 2006.

[4] A. Charfi, and M. Mezini, “Ao4bpel: An aspect-oriented extension to bpel,” World Wide Web, vol. 10, no. 3, pp. 309-344, 2007.

[5] R. Elmasri, and S. B. Navathe, Fundamentals of database systems (6th ed.): Addison-Wesley Longman Publishing Co., Inc., 2010.

[6] H. Garcia-Molina, and K. Salem, “Sagas,” ACM SIGMOD Record, vol. 16, no. 3, pp. 249-259, 1987.

[7] J. Jang, A. Fekete, and P. Greenfield, “Delivering Promises for Web Services Applications,” IEEE International Conference on Web Services, Salt Lake City, Utah, USA, 2007.

[8] Y. Jin, “An architecture and execution environment for component integration rules,” PhD Dissertation, Arizona State University, 2004.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, “An overview of AspectJ,” ECOOP 2001—Object-Oriented Programming, pp. 327-354, 2001.

[10] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of JML: A behavioral interface specification language for Java,” ACM SIGSOFT Software Engineering Notes, vol. 31, no. 3, pp. 1-38, 2006.

[11] D. Luckham, Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs: Springer New York, 1990.

[12] A. Martens, “Analyzing web service based business processes,” Fundamental Approaches to Software Engineering, pp. 19-33, 2005.

[13] T. Mikalsen, S. Tai, and I. Rouvellou, "Transactional attitudes: Reliable composition of autonomous Web services." Workshop on Dependable Middleware Based Systems, 2002

[14] R. Shrestha, “Using Assurance Points and Integration Rules for Recovery in Service Composition,” MS Thesis, Texas Tech University, 2010.

[15] M. Shuman, *A Database Service for Checking Invariants*. Technical Report, Department of Computer Science, Texas Tech University, 2010.

[16] M. Tumma, Oracle Streams: High Speed Replication and Data Sharing: Rampant TechPress, 2004.

[17] S. D. Urban, S. W. Dietrich, Y. Na, Y. Jin, A. Sundermier, and A. Saxena, "The IRules Project: Using Active Rules for the Integration of Distributed Software Components." 9th IFIP Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems, pp. 265-286, 2001.

[18] S. D. Urban, Y. Xiao, L. Blake and S. W. Dietrich, “Monitoring data dependencies in concurrent process execution through delta-enabled grid services,” International Journal of Web and Grid Services, vol. 5, no. 1, pp. 85-106, 2009.

[19] S. Urban, L. Gao, R. Shrestha and A. Courter, “Achieving Recovery in Service Composition with Assurance Points and Integration Rules,” On the Move to Meaningful Internet Systems: OTM 2010, pp. 428-437, 2010.

[20] H. Wächter, and A. Reuter, The contract model: Advanced Transaction Models for New Applications, Morgan Kaufmann Publishers, 1991.

[21] D. Worah, and A. Sheth, "Transactions in transactional workflows," Advanced Transaction Models and Architectures, S. J. a. L. Kershberg, ed., pp. 3-34., 1997.

[22] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, “A reservation-based coordination protocol for Web Services,” in IEEE International Conference on Web Services, Orlando, Florida, 2005.